MEMORIA ESTÁTICA Y DINÁMICA EN C

Uso de malloc, calloc, realloc y free

Tema	Apuntadores, memoria y gestión dinámica
Nivel	Programación en C - intermedio / avanzado inicial
Propósito	Comprender cómo reservar, utilizar, redimensionar y liberar memoria durante la ejecución

Idea central

La memoria dinámica permite que un programa solicite espacio durante su ejecución, almacene allí datos mediante apuntadores y libere ese espacio cuando deja de necesitarlo.

Guía con explicaciones conceptuales, diagramas ASCII, ejemplos completos, buenas prácticas y errores frecuentes.

Contenido

1. Memoria en un programa C

Cuando un programa se ejecuta necesita espacio para almacenar variables, arreglos, estructuras, cadenas y otros datos temporales. En C, el programador puede trabajar tanto con objetos cuyo tamaño se conoce de antemano como con bloques obtenidos durante la ejecución.

> Concepto fundamental: una variable no solo tiene un valor; también ocupa una posición en memoria. La dirección de esa posición puede obtenerse con el operador & y almacenarse en un apuntador.

```
int edad = 20;

printf("Valor: %d\n", edad);
printf("Direccion: %p\n", (void *)&edad);
```

Visualización conceptual:

```
             memoria
               ↓
          +---------------+
          |      20       |  ← valor
          +---------------+
                 ↑
                 |
          direccion de edad
```

La dirección exacta cambia entre ejecuciones y sistemas. Por eso, al enseñar el concepto, conviene hablar de “una dirección” o de un ejemplo hexadecimal, sin asumir que siempre tendrá el mismo valor.

La memoria dinámica añade otra posibilidad: el programa puede pedir un bloque de memoria en tiempo de ejecución y obtener una dirección que luego se guarda en un apuntador.

2. Memoria estática y automática

En material introductorio suele llamarse memoria estática o de tamaño determinado a los objetos cuyo espacio se establece en la declaración o por reglas de almacenamiento del lenguaje, sin que el programador tenga que solicitar un bloque con malloc o calloc.

```
int edad;
float promedio;
char nombre[30];
```

En el caso de char nombre[30], el vector tiene espacio para 30 caracteres. No es posible cambiar ese mismo objeto para convertirlo en un vector de 100 elementos mediante una simple asignación.

Ejemplo de vector de tamaño fijo:

```
int numeros[5];

numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
```

Este enfoque es sencillo y seguro en muchos programas pequeños, pero resulta menos flexible cuando el número de elementos solo se conoce durante la ejecución.

Distinción importante: en el modelo de C existen varios tipos de almacenamiento (automático, estático, dinámico, etc.). Para una enseñanza inicial, es mejor no afirmar que toda variable “vive en la memoria estática”; por ejemplo, una variable local ordinaria suele tener almacenamiento automático. Lo esencial para este tema es contrastar la gestión automática con la gestión dinámica explícita.

3. Memoria dinámica: concepto y propósito

La memoria dinámica es un área que el programa solicita durante la ejecución. El tamaño puede depender de datos que todavía no se conocían al compilar el programa.

Supongamos que el usuario responde:

```
¿Cuantos estudiantes hay? 35
```

El programa puede reservar exactamente el espacio necesario:

```
int n;
scanf("%d", &n);

int *notas = malloc(n * sizeof(int));
```

El flujo conceptual es:

```
dato del usuario
      ↓
calcular tamaño
      ↓
solicitar memoria
      ↓
recibir una direccion
      ↓
guardar la direccion en un apuntador
      ↓
usar el bloque
```

La ventaja principal es la flexibilidad: el programa puede adaptar el tamaño de la estructura a las necesidades reales de la ejecución.

4. malloc: reserva de memoria

malloc pertenece a la biblioteca estándar stdlib.h y solicita un bloque de memoria de un tamaño especificado en bytes.

```
#include <stdlib.h>

int *p;
p = malloc(sizeof(int));
```

El resultado de malloc es una dirección de memoria que se guarda en p. Si la solicitud falla, malloc devuelve NULL.

```
int *p = malloc(sizeof(int));

if (p == NULL)
{
    printf("No se pudo reservar memoria.\n");
    return 1;
}
```

Después de reservar, el espacio debe inicializarse antes de leerse como un dato válido:

```
*p = 25;
printf("%d\n", *p);
```

Importante: malloc no inicializa el contenido del bloque. El contenido inicial no debe tratarse como si fuera un valor conocido.

Uso de sizeof

Se recomienda usar sizeof(tipo) en lugar de asumir un tamaño en bytes. Por ejemplo:

```
int *p = malloc(sizeof(int));
float *f = malloc(sizeof(float));
double *d = malloc(sizeof(double));
```

Para reservar un vector de n elementos:

```
int *numeros = malloc(n * sizeof(int));
```

La expresión calcula automáticamente el número de bytes necesarios para almacenar n objetos del tipo int.

5. calloc: reserva e inicialización en cero

calloc también pertenece a stdlib.h, pero recibe dos argumentos: cantidad de elementos y tamaño de cada elemento.

```
int *numeros = calloc(5, sizeof(int));
```

Una diferencia didáctica esencial frente a malloc es que calloc inicializa el bloque reservado a cero (es decir, todos sus bytes quedan en cero). Para tipos enteros habituales, esto produce valores 0.

Visualización:

```
malloc(5 * sizeof(int))

+----+----+----+----+----+
| ?  | ?  | ?  | ?  | ?  |
+----+----+----+----+----+

calloc(5, sizeof(int))

+----+----+----+----+----+
|  0 |  0 |  0 |  0 |  0 |
+----+----+----+----+----+
```

Ejemplo completo:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int n = 5;
    int *numeros = calloc(n, sizeof(int));

    if (numeros == NULL)
    {
        printf("Error de memoria.\n");
        return 1;
    }

    for (int i = 0; i < n; i++)
        printf("%d ", numeros[i]);

    free(numeros);
    return 0;
}
```

Salida esperada:

```
0 0 0 0 0
```

Use calloc cuando el algoritmo se beneficie de iniciar el bloque con cero. No es obligatorio usar calloc para toda reserva dinámica.

6. realloc: cambiar el tamaño de un bloque

realloc modifica el tamaño de un bloque obtenido anteriormente con malloc, calloc o realloc. Puede ampliar o reducir el espacio.

```
int *numeros = malloc(5 * sizeof(int));

/* mas adelante... */
int *temp = realloc(numeros, 8 * sizeof(int));
```

La memoria puede permanecer en la misma ubicación o ser trasladada a otra zona del sistema. Por eso nunca conviene suponer que la dirección se conservará.

Visualización conceptual:

```
ANTES

     numeros
        ↓
+----+----+----+----+----+
| 10 | 20 | 30 | 40 | 50 |
+----+----+----+----+----+

DESPUES

     numeros
        ↓
+----+----+----+----+----+----+----+----+
| 10 | 20 | 30 | 40 | 50 | ?  | ?  | ?  |
+----+----+----+----+----+----+----+----+
```

Forma segura de usar realloc

```
int *temp = realloc(numeros, nuevo_tamano);

if (temp == NULL)
{
    /* numeros original sigue disponible */
    printf("No se pudo redimensionar.\n");
}
else
{
    numeros = temp;
}
```

¿Por qué no hacer siempre numeros = realloc(numeros, ...)? Porque, si realloc falla y devuelve NULL, se perdería la referencia al bloque original. Con un apuntador temporal se puede manejar el error sin perder esa referencia.

Ejemplo de ampliación de un vector:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int n = 5;
    int nuevo_n = 8;

    int *numeros = malloc(n * sizeof(int));
    if (numeros == NULL) return 1;

    for (int i = 0; i < n; i++)
        numeros[i] = (i + 1) * 10;

    int *temp = realloc(numeros, nuevo_n * sizeof(int));

    if (temp == NULL)
    {
        free(numeros);
        return 1;
    }

    numeros = temp;

    for (int i = n; i < nuevo_n; i++)
        numeros[i] = 0;

    for (int i = 0; i < nuevo_n; i++)
        printf("%d ", numeros[i]);

    free(numeros);
    return 0;
}
```

Los elementos que ya existían se conservan dentro de las garantías de realloc mientras el proceso es exitoso; las posiciones nuevas, cuando se amplía, no deben tratarse como inicializadas a cero automáticamente. El programa debe establecer sus valores antes de usarlas.

7. free: liberar memoria

free informa al sistema que un bloque de memoria dinámica ya no se necesita y puede reutilizarse.

```
int *p = malloc(sizeof(int));

*p = 25;

free(p);
```

Después de liberar un bloque, no debe accederse a él mediante ese apuntador como si siguiera vigente.

```
free(p);

/* Incorrecto: */
printf("%d", *p);
```

Una práctica recomendable después de liberar:

```
free(p);
p = NULL;
```

Asignar NULL no “repara” la memoria ni sustituye a free. Su función es ayudar a que el apuntador deje de contener una dirección que ya no debería utilizarse.

El ciclo de vida básico queda así:

```
malloc / calloc
       ↓
    MEMORIA
       ↓
    UTILIZAR
       ↓
  realloc (si es necesario)
       ↓
    UTILIZAR
       ↓
      free
       ↓
MEMORIA LIBERADA
```

8. Fugas de memoria y punteros colgantes

Una fuga de memoria (memory leak) ocurre cuando el programa reserva memoria y pierde la referencia al bloque sin liberarlo. El bloque queda ocupado, pero el programa ya no sabe cómo llegar a él.

```
int *p = malloc(100 * sizeof(int));

p = malloc(200 * sizeof(int));
```

En este ejemplo se perdió la dirección del primer bloque. La segunda asignación sobreescribió la única referencia que se tenía a esa memoria.

Un puntero colgante (dangling pointer) aparece cuando un apuntador conserva la dirección de un bloque cuya memoria ya fue liberada.

```
int *p = malloc(sizeof(int));
*p = 10;

free(p);

/* p todavía puede contener una dirección,
   pero ya no debe usarse para acceder al bloque. */
```

La práctica free(p); p = NULL; ayuda a reducir el riesgo de reutilizar accidentalmente un puntero colgante.

También debe evitarse liberar dos veces el mismo bloque:

```
free(p);
free(p);   /* incorrecto */
```

La doble liberación produce comportamiento indefinido. Antes de liberar, es válido comprobar si el apuntador es distinto de NULL.

9. Comparación completa de las cuatro funciones

Función	Propósito	Inicialización	Forma típica
malloc	Reserva un bloque de bytes	No inicializa el contenido	malloc(n * sizeof(T))
calloc	Reserva un arreglo de elementos	Inicializa todos los bytes a cero	calloc(n, sizeof(T))
realloc	Cambia el tamaño de un bloque dinámico	No “pone en cero” las nuevas posiciones por sí mismo	realloc(p, nuevo_tam)
free	Libera un bloque dinámico	No aplica	free(p)

Regla mnemotécnica:

```
malloc  → crear espacio
calloc  → crear espacio en cero
realloc → cambiar tamaño
free    → liberar espacio
```

10. Memoria dinámica con vectores

La aplicación más común en cursos iniciales es construir un vector cuyo tamaño se conoce solamente durante la ejecución.

```
int n;
printf("Cantidad de datos: ");
scanf("%d", &n);

int *datos = malloc(n * sizeof(int));
```

La dirección inicial se almacena en datos. Por ello, el vector puede utilizarse con índices:

```
datos[0] = 10;
datos[1] = 20;
datos[2] = 30;
```

La relación con los apuntadores es importante: el nombre de un arreglo, en la mayoría de expresiones, se convierte en un apuntador al primer elemento. Por eso, para una variable v de tipo arreglo, v[i] es conceptualmente equivalente a *(v + i).

```
datos[i]      ≡      *(datos + i)
```

Ejemplo completo: vector dinámico de notas

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int n;

    printf("Cantidad de estudiantes: ");
    scanf("%d", &n);

    if (n <= 0)
    {
        printf("Cantidad no valida.\n");
        return 1;
    }

    float *notas = malloc(n * sizeof(float));

    if (notas == NULL)
    {
        printf("No se pudo reservar memoria.\n");
        return 1;
    }

    float suma = 0.0f;

    for (int i = 0; i < n; i++)
    {
        printf("Nota %d: ", i + 1);
        scanf("%f", &notas[i]);
        suma += notas[i];
    }

    printf("Promedio = %.2f\n", suma / n);

    free(notas);
    notas = NULL;

    return 0;
}
```

Aquí se integran variables, apuntadores, vectores y memoria dinámica. El usuario decide el tamaño, el programa reserva exactamente ese número de elementos y al final libera el bloque.

11. Memoria dinámica y funciones

Los bloques dinámicos pueden pasarse a funciones mediante apuntadores. En C, una función puede recibir la dirección inicial del vector y trabajar sobre sus elementos.

```
void mostrar(const int *v, int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", v[i]);
}
```

La palabra const indica que la función solo debe leer el vector mediante ese apuntador. La memoria sigue siendo la misma; lo que cambia es la restricción sobre su modificación a través de v.

Una función puede modificar el bloque original si recibe un apuntador no constante:

```
void llenar(int *v, int n)
{
    for (int i = 0; i < n; i++)
        v[i] = (i + 1) * 5;
}
```

Uso:

```
int *vector = malloc(5 * sizeof(int));

if (vector != NULL)
{
    llenar(vector, 5);
    mostrar(vector, 5);
    free(vector);
    vector = NULL;
}
```

Una función incluso puede crear un bloque y devolver su dirección:

```
int *crearVector(int n)
{
    int *v = malloc(n * sizeof(int));
    return v;
}
```

En ese caso, la función que recibe el resultado debe asumir la responsabilidad de liberar la memoria cuando termine de utilizarla.

También es posible devolver varios resultados mediante parámetros apuntadores:

```
void operaciones(int a, int b, int *suma, int *producto)
{
    *suma = a + b;
    *producto = a * b;
}

int suma, producto;
operaciones(5, 3, &suma, &producto);
```

Esto conecta el manejo dinámico con el tema anterior de paso por valor y modificación mediante apuntadores.

12. Ejemplo integrador

El siguiente programa reúne reserva, validación, uso de funciones, redimensionamiento y liberación.

```
#include <stdio.h>
#include <stdlib.h>

void llenar(int *v, int n)
{
    for (int i = 0; i < n; i++)
        v[i] = (i + 1) * 10;
}

void mostrar(const int *v, int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
```

```
}

int main(void)
{
    int n = 5;
    int nuevo_n = 8;

    int *vector = malloc(n * sizeof(int));

    if (vector == NULL)
        return 1;

    llenar(vector, n);

    printf("Vector inicial: ");
    mostrar(vector, n);

    int *temp = realloc(vector, nuevo_n * sizeof(int));

    if (temp == NULL)
    {
        free(vector);
        return 1;
    }

    vector = temp;

    for (int i = n; i < nuevo_n; i++)
        vector[i] = 0;

    printf("Vector ampliado: ");
    mostrar(vector, nuevo_n);

    free(vector);
    vector = NULL;

    return 0;
}
```

Posible salida:

```
Vector inicial: 10 20 30 40 50
Vector ampliado: 10 20 30 40 50 0 0 0
```

Observe el orden lógico:

```
malloc → reservar
llenar → utilizar
realloc → redimensionar
mostrar → utilizar
free → liberar
NULL → marcar que ya no apunta a un bloque válido
```

Este ciclo de vida es un patrón central de la programación con memoria dinámica.

13. Errores frecuentes y buenas prácticas

1. Olvidar incluir stdlib.h

```
#include <stdio.h>
#include <stdlib.h>
```

2. No verificar el resultado de malloc/calloc/realloc

```
int *p = malloc(n * sizeof(int));
if (p == NULL) { /* manejar error */ }
```

3. Leer memoria de malloc antes de inicializarla

```
int *p = malloc(sizeof(int));
/* no leer *p todavía */
*p = 10;
```

4. Perder una referencia antes de liberar

```
int *p = malloc(100 * sizeof(int));
p = malloc(200 * sizeof(int));  /* fuga del primer bloque */
```

5. Usar memoria después de free

```
free(p);
/* no usar *p */
```

6. Liberar dos veces

```
free(p);
free(p);  /* comportamiento indefinido */
```

7. Usar directamente el resultado de realloc sobre el mismo apuntador cuando se necesita conservar la referencia original en caso de fallo

```
int *temp = realloc(p, nuevo_tamano);
if (temp != NULL)
    p = temp;
```

8. Suponer que realloc siempre conserva la dirección

realloc puede devolver la misma dirección o una diferente. El programador debe trabajar con el nuevo valor devuelto si la operación tuvo éxito.

9. No controlar tamaños inválidos

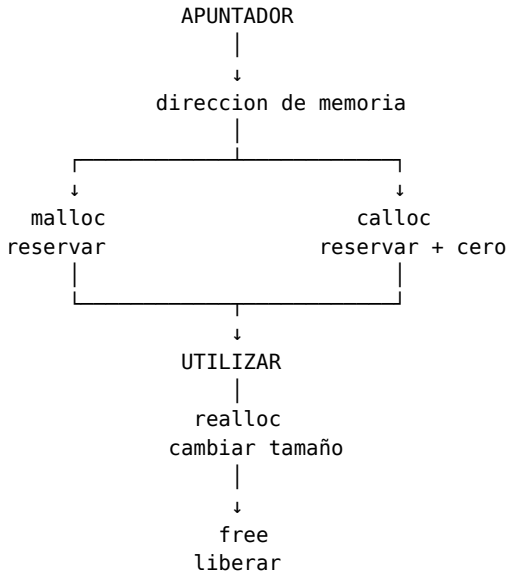
```
if (n <= 0)
    return 1;
```

10. Ignorar la gestión de la vida útil del bloque

Toda reserva dinámica debe asociarse mentalmente a una responsabilidad: “¿quién la libera y en qué momento?”.

14. Resumen y reglas de oro

Mapa conceptual:



Las reglas esenciales para el estudiante son:

- malloc/calloc reservan memoria dinámica.
- malloc no inicializa el bloque; calloc lo deja con todos los bytes en cero.
- realloc cambia el tamaño y puede mover el bloque a otra dirección.
- free libera el bloque dinámico cuando ya no se necesita.
- Siempre se debe manejar la posibilidad de que una reserva falle y devuelva NULL.
- Después de free, no se debe volver a acceder al bloque liberado.
- Una práctica útil es hacer p = NULL después de liberar el bloque.
- La memoria dinámica es responsabilidad del programador: reservar, usar correctamente y liberar.

Fórmula mental:

Reservar → comprobar → inicializar → utilizar → redimensionar si hace falta → liberar.

15. Ejercicios propuestos

Los ejercicios están diseñados para que el estudiante compruebe si entiende el ciclo de vida de la memoria y no solo la sintaxis.

Ejercicio 1 - Vector dinámico básico

Solicite al usuario n, reserve un vector de enteros con malloc, lea n valores, muestre el mayor y libere el bloque.

Ejercicio 2 - calloc y conteo

Cree un vector de 10 contadores con calloc. Lea 20 números enteros entre 0 y 9 y cuente cuántas veces aparece cada número.

Ejercicio 3 - crecer un vector

Comience con capacidad 2. Agregue números uno por uno y, cuando la capacidad se llene, use realloc para duplicarla. Al final muestre todos los valores y libere la memoria.

Ejercicio 4 - detectar errores

```
int *p = malloc(5 * sizeof(int));
free(p);
free(p);
printf("%d", *p);
```

Identifique todos los problemas del fragmento y proponga una versión segura.

Ejercicio 5 - función que crea memoria

Escriba una función que reciba n, cree un vector dinámico, lo inicialice con los números del 1 al n y devuelva la dirección del vector. En main, muestre los datos y libere la memoria.

Ejercicio 6 - razonamiento sin ejecutar

```
int *p = calloc(4, sizeof(int));
p[0] = 7;
p[1] = 9;

int *q = realloc(p, 6 * sizeof(int));
```

Explique qué información se conserva, qué posiciones nuevas deben considerarse no inicializadas para los propósitos del algoritmo, qué puede ocurrir con la dirección y por qué conviene comprobar q antes de reemplazar p.

Pregunta final de comprensión

¿Por qué un programa con memoria dinámica necesita pensar no solamente en “dónde guardo los datos”, sino también en “quién es responsable de liberar ese espacio y cuándo”?

Apéndice - Comandos esenciales

Operación	Sintaxis	Idea clave
Declarar apuntador	int *p;	p almacenará una dirección compatible con int
Obtener dirección	&x;	dirección de x
Dereferenciar	*p	valor ubicado en la dirección almacenada en p
Reservar	p = malloc(n * sizeof(int));	memoria dinámica sin inicialización garantizada
Reservar en cero	p = calloc(n, sizeof(int));	memoria dinámica con bytes en cero
Redimensionar	temp = realloc(p, nuevo);	cambia tamaño; puede mover el bloque
Liberar	free(p);	deja el bloque disponible para reutilización
Anular apuntador	p = NULL;	evita conservar una dirección que ya no debe usarse

Nota de precisión

En C, “paso por referencia” no es una categoría formal del lenguaje como en otros lenguajes. Cuando una función puede modificar una variable del llamador, normalmente se le pasa la dirección mediante un apuntador. Esta guía usa “mediante apuntador” para describir ese mecanismo con precisión.

Fin del material.